,

# RISC Performance Improvements to the LAURA Code

Robert J. Bergeron

Report NAS-00-010 January 2000

## Abstract

As supercomputing workloads transition from vector to RISC architectures, vectorized CFD algorithms must adapt to execution on microprocessors. The initial execution of these algorithms on a RISC microprocessor will be typically memory-bound because vectorized floating point calculations require more data than the microprocessor cache can deliver. Since many of these applications execute efficiently on vector machines, algorithmic improvements for RISC machines may not be obvious. This paper presents performance data resulting from some simple algorithm modifications to an efficiently vectorized NAS workload code. The modifications involved both replacing the "sliding-window" technique by an approach which placed the entire problem in core, and consolidating several subroutines to decrease overhead. The modifications increased floating point performance by about 15%.

## 1. Introduction

The introduction of RISC computers into the NAS environment will require the vectorized NAS algorithms to adapt to microprocessor architectures. The typical NAS code executing on the vector C90 machines achieves about 25% of peak performance[1], but NAS workloads executing on RISC platforms display efficiencies of 3-7%[2]. This lower level of efficiency has motivated a desire to improve performance by adapting the existing codes to the microprocessor environment. While one way to improve performance may be to utilize more processors through parallelism, this report emphasizes single-CPU performance because many user codes do not scale beyond some limited number of processors, i.e., adding more processors beyond this limit does not significantly reduce elapsed time. The code algorithms may scale, but the code applications do not.

Because the latency and bandwidth of the RISC microprocessor memory systems have not kept pace with raw processor speed, many of the NAS computational fluid dynamics (CFD) computer codes are memory bound. Optimization requires both an algorithmic implementation and a coding style which balances memory references and floating point operations in order to minimize the amount of cycles the processor spends waiting for data from memory. Since adapting existing codes generally precludes choosing a substantially new algorithm, optimization usually reduces to redoing the original programming style.

Adapting a code to the RISC architecture can include a variety of activities beginning with unrolling outer loops and merging the resulting copies of inner loops[3]. Typically, the innermost loop of a vector code operates on the largest dimension of a multidimensional array. Reordering the indices of this loop and those of the array to allow this loop to execute on "in-cache" data is another popular optimization. Blocking loops for cache, i.e., processing loop data in cache-sized chunks, can improve performance if the loop processes the data arrays with large stride or if the loop reuses data values many times. However, a code executing efficiently on a vector computer may already be employing good programming practices and the usual prescriptions for improving performance may not apply.

The LAURA (Langley Aerothermodynamics Upwind Relaxation Algorithm) code[4] is an efficient vector

code which plays a prominent role in the NAS supercomputing workload. LAURA displays a C-90 performance of about 300 Mflops/CPU. Code authors kindly provided a recent source code[4], version 4.5, and a small sample problem, as a template to investigate what sort of changes could lead to higher CPU efficiency.
This report discusses the changes made to the LAURA code to increase RISC CPU performance by 15% on the SGI Origin2000. The principal goal of this report is an explanation of the process required to modify an algorithm, designed to run well on one architecture, to run even better on a second architecture. Such a modification requires some understanding of both the algorithm and the architecture. Section 2 describes the LAURA algorithm and data structure. Section 3 discusses the initial LAURA performance. Section 4 discusses the optimization and Section 5 presents some conclusions.

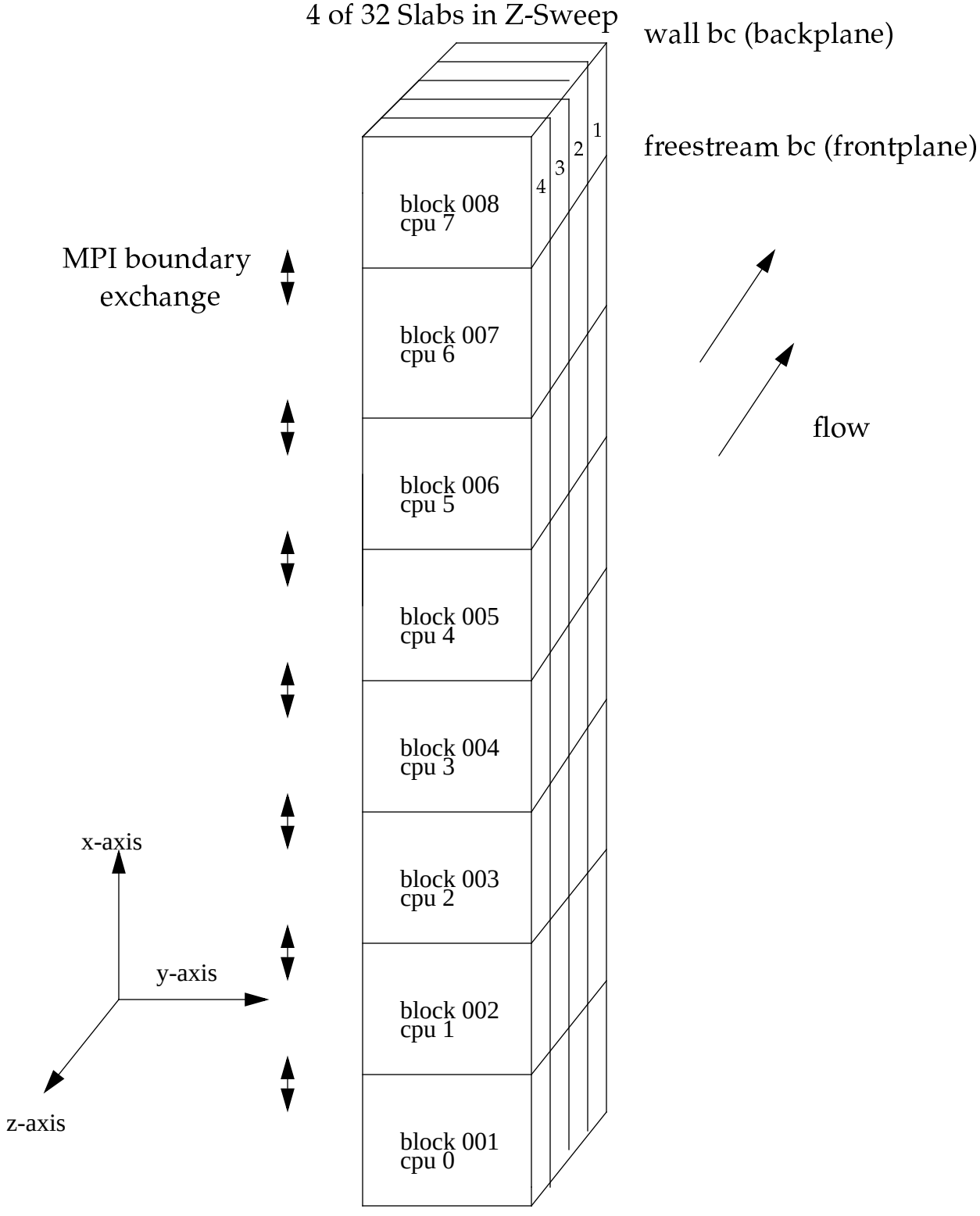
## 2. Description of the LAURA Code

LAURA treats hypersonic, viscous flow around space transfer vehicles as well as flow within supersonic scramjet engines and nozzles. The code employs a finite volume, shock-capturing algorithm to simulate viscous or inviscid steady-state flows. LAURA employs an upwind-biased, flux-difference splitting algorithm with second-order corrections based on a total-variation-diminishing (TVD) scheme. In contrast to the fully factored treatments [5], which sequentially solve systems of block tridiagonal equations, LAURA employs a "point implicit" strategy which solves a system consisting of simultaneous linear equations with a simple Gaussian technique. Slabs adjoining a given slab in the non-sweep direction (i.e., in the x- and y-direction for a z-sweep) propagate the changes in their face values via boundary exchanges in an asynchronous fashion to allow the iteration to use the latest available data. The LAURA package provides both shared memory and MPI (Message-Passing Interface) versions; the current effort used the MPI version.

The MPI version of LAURA decomposes the computational domain into blocks with each block assigned to a processor. Figure 1 shows an idealized picture of the z-slabs and z-sweep for the 8-block sample problem. Of the 32 z-slabs in the sample problem, the figure shows slabs 1 to 4 with slab 1 adjacent to the wall and slab 4 a distance away from the wall; slab 32 is adjacent to the freestream boundary. Each processor traverses the rectangular slabs comprising its block against the flow direction, beginning with the slab nearest the vehicle surface and its wall boundary condition, and moving across the boundary layer out to the freestream end of the mesh. The code then reverses the sweep, visiting the slabs in the direction of the flow toward the slab on the vehicle surface. Upon visiting a slab, the code computes gradients of the characteristic variables at the z-interfaces, limits these gradients according to the TVD scheme, relaxes the governing equations to form the left-hand side, and solves for the changes in the flow variables. The vertical arrows in the figure show MPI boundary data exchanges for those faces having a common interface. These boundary exchanges occur at user-specified intervals and time spent in MPI execution does not contribute significantly to the elapsed time. The figure also shows that front and back faces lie in the direction of the flow and that the other four sides are normal to the flow. Since the blocks all contain the same number of nodes, this problem is well-balanced for parallel execution. Further, the memory required for each block fits within a single CPU, isolating each CPU's memory loads and stores from those of the other CPUs. Thus, this sample problem provides an excellent vehicle for investigating single CPU performance.

LAURA is highly vectorized. The programming approach collapses the two dimensions associated with the computational plane into a single FORTRAN loop index to give longer vectors. Since typical large problems can involve millions of slabs, the algorithm employs a sliding window technique to keep only several (3 for first-order spatial accuracy, 5 for second-order spatial accuracy) of the slabs in the working memory for any slab in the sweep. The small size of the early vector machine memories necessitated this approach and the memory speed of the current generation of vector computers allows this technique to produce a high floating-point performance rate, even though the windowing technique places a strong demand on the data transfer rate of the memory subsystem.

Figure 1
Structure of LAURA 8-block Sample Problem



Although LAURA is highly vectorized and runs well on the C90, it may run poorly on a RISC machine. The first way to identify possible inefficiencies is to examine where the code is spending its time. Table 1 quantifies the time spent in the various subroutines and provides the percentages of total elapsed time contributed by the major subroutines.

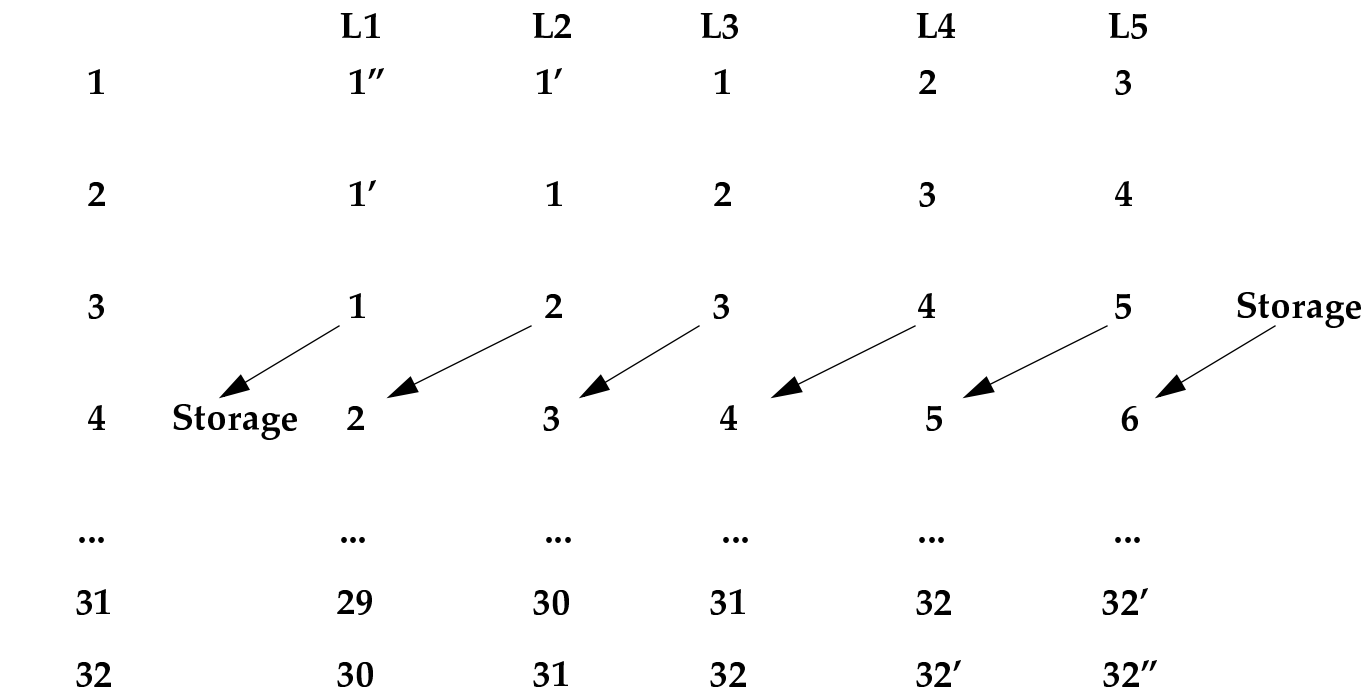
**Table 1: Initial LAURA Subroutine Elapsed Times-1000 Steps**

| Subroutine | Elapsed Time | Percentage | Comment |
|---|---|---|---|
| gatface | 230.2 | 20.8 | place central defined-slab data into slab face arrays |
| drv | 151.5 | 13.7 | Roe averaging of characteristic variables |
| invflx | 100.4 | 9.1 | compute inviscid component of flux: |
| dirswp | 97.43 | 8.8 | manage slab data placement and direct the sweep along the slabs |
| gatgeoa | 94.79 | 8.5 | gather geometry data for the current slab into the current face |
| minmod | 94.37 | 8.5 | calculate minmod of two vectors, element by element |
| dropone | 74.91 | 6.8 | move slab data from one local slab to another local slab in the sliding window |
| gatscta | 59.47 | 5.4 | gather slab data from local slab 3 into the global array or scatter data from the global array into the local array. |
| limiter | 32.30 | 2.9 | limits changes in the characteristic variables via minmod |
| total | 1109.0 | 1.000 | total elapsed time |

The table shows that much of the code involves loading slab data into various faces and returning slab data into permanent storage arrays. The table identifies subroutines GATFACE, GATGEOA, DROPONE, and GATSCTA as accounting for over 40% of the elapsed time. These routines do very little computation. Instead, they comprise the main routines implementing the sliding window technique.

Since the sliding window technique accounts for much of the elapsed time, an understanding of this method is important to see whether modification can reduce the elapsed time involved. Figure 2, illustrating the local and global numbering employed in this technique, shows both the placement of the global slab data in the sliding window and the required data movement as the code visits the slabs. Global numbering is the same as that shown in Figure 1: there are 32 slabs in the z-sweep and slab 1 lies adjacent to the wall and slab 32 lies adjacent to the freestream. Local numbering refers to the computational scheme. Second-order accuracy requires 5 local slabs, two downstream(L1 and L2) and two upstream(L4 and L5) of the working slab (L3). To compute global slab 1(adjacent to the wall), data for pseudo-boundary slabs 1" and 1’ must be placed into local slabs L1 and L2. The data for global slab 1 is placed into local slab L3. Placement of global slab 2 data and global slab 3 data into local slab L4 and L5 completes the sliding window. The global slab 1 computation is now ready to proceed. The arrows show

**Figure 2**
**LAURA Sliding Window Technique**
**For Global Slabs 1-32 of Block 1**




**To compute global slab 1 with second-order accuracy, LAURA requires that:**
**Local slab L1 must contain data for a pseudo-boundary slab 1″**
**Local slab L2 must contain data for a pseudo-boundary slab 1′**
**Local slab L3 must contain data for global slab 1**
**Local slab L4 must contain data for global slab 2**
**Local slab L5 must contain data for global slab 3**

**To compute global slab 32 with second-order accuracy, LAURA requires that:**
**Local slab L1 must contain data for global slab 30**
**Local slab L2 must contain data for global slab 31**
**Local slab L3 must contain data for global slab 32**
**Local slab L4 must contain data for a pseudo-boundary slab 32′**
**Local slab L5 must contain data for a pseudo-boundary slab 32″**

the data movement required somewhat later in the z-sweep just before the calculation of global slab 4 and represent the following sequence and subroutines:

Storing local slab 1 into the global array (GATSCTA)
Moving local slab 2 (global slab 2) into local slab 1(DROPONE)
Moving local slab 3 (global slab 3) into local slab 2(DROPONE)
Moving local slab 4 (global slab 4) into local slab 3(DROPONE)
Moving local slab 5 (global slab 5) into local slab 4(DROPONE)
Retrieving global slab 6 from global array into local slab 5 (GATSCTA)

## 3. LAURA Performance on SGI Origin

The R10000 is a superscalar RISC processor, capable of fetching and decoding 4 instructions per cycle which can execute on its 5 independent pipelined functional units: a load/store unit, 2 arithmetic logic units, a floating-point add unit, and a pipelined floating point multiply unit. The latter two units can be chained together to perform multiply-add instructions. The R10000 provides a two-level cache hierarchy, a 32 KB 2-way set associative level 1 on-chip cache with a 64 byte line size and a 4 MB 2-way set associative Level 2 off-chip secondary cache with a 128 byte line size. The time to access data, the memory latency, is 2 or 3 clock cycles for data in the L1 cache and 8 to 10 clock cycles for data in the L2 cache. If the processor has to access data in user main memory, i.e., the data is not already in the primary or secondary cache, the latency is at least 60 clock cycles. The Origin employs a virtual memory Operating System (OS) which arranges user main memory into areas called pages. If the desired data does not reside in the current pageset, a TLB (Translation Lookaside Buffer) miss occurs as the OS must bring in the page containing the desired data. While there are different delays associated with the various levels of TLB misses, the TLB delay experienced by LAURA is about 70 cycles per miss.

The R10000 processor supplies two counters, termed Event Counters (EC) for reporting certain hardware events and the information provided by these counters was extremely helpful in deciding how this processor limited the LAURA code performance. Each of the R10000 counters can track one event at a time and provides a choice of 16 events per counter. There are also two associated control registers which are used to specify which event the relevant counter is counting. Each counter is a 32-bit read / write register and is incremented by one each time the event specified in its associated control register occurs. *Perfex*, the utility which reads the event counters, can execute in one of two modes. In sampling mode, *Perfex* samples all of the reported hardware events and thus can provide approximate counts for 32 events. In exact mode, *Perfex* provides exact counts for 2 user-specified events and a complete performance picture would require 16 executions. *Perfex* reports raw R10000 event counts and associates an approximate time cost with the event counts [7].

*Perfex* also reports a number of statistics derived from the typical time costs, and Table 2 describes five key memory statistics.

**Table 2: Perfex-derived Memory Statistics**

| Statistic | Description |
|---|---|
| L1 Cache Line Reuse | Average number of times that a primary data cache line is used after it has been moved into the cache. |
| L2 Cache Line Reuse | Average number of times that a secondary data cache line is used after it has been moved into the cache. |

**Table 2: Perfex-derived Memory Statistics**

| Statistic | Description |
|---|---|
| L1 Data Cache Hit Rate | Fraction of data accesses which are satisfied from a cache line already resident in the primary data cache. |
| L2 Data Cache Hit Rate | Fraction of data accesses which are satisfied from a cache line already resident in the secondary data cache. |
| Time accessing memory/ Total time | Total of the typical costs of graduated loads, graduated stores, primary data cache misses, secondary data cache misses, and TLB misses divided by the total program run time. |

Table 3 contains the key *Perfex* reported events ranked in order of the highest time cost for the original LAURA. The largest costs occur for the primary (on-chip L1) data cache misses and memory loads. Maximum performance occurs when the processor operates on data in its primary cache, and the decrease in floating point performance when the processor operates on data in its secondary cache is about 30% for ideal loops [8]. *Perfex* output for LAURA execution indicates that secondary cache and TLB misses make smaller contributions to the overall execution costs. *Perfex* statistics confirm the relatively good use of the cache memory hierarchy with 91 primary and 99% secondary cache memory hit rates. *Perfex* reports a single processor performance rate of 35 MFLOPS. This rate is a underestimate because *Perfex* cannot distinguish floating-point multiply-add from a floating point multiply or floating point add. Subsequent executions of LAURA in exact mode indicated that the correct rate was 45 MFLOPS. *Perfex* statistics also report that a floating point operation requires over 2 memory operations, significantly higher than the corresponding ratio required for the standard NAS workload suites.

**Table 3: Selected SGI Event Counters -Initial LAURA-300 Steps**

| Reported Event | Estimated Event Cost (sec) |
|---|---|
| Cycles | 921.3 |
| Primary data cache misses | 334.2 |
| Memory Loads | 274.5 |
| Floating point instructions | 164.2 |
| Memory stores | 137.1 |
| Quadwords written back from primary data cache | 132.6 |
| Decoded branches | 58.2 |
| Secondary data cache misses | 37.7 |
| Quadwords written back from scache | 12.3 |
| TLB misses | 9.6 |
| | |

**Table 3: Selected SGI Event Counters -Initial LAURA-300 Steps**

| Reported Event | Estimated Event Cost (sec) |
|---|---|
| **Statistics** | |
| Graduated loads & stores/floating point instruction | 2.2 |
| L1 Cache Line Reuse | 10.0 |
| L2 Cache Line Reuse | 73.2 |
| L1 Data Cache Hit Rate | 0.909 |
| L2 Data Cache Hit Rate | 0.986 |
| Time accessing memory/Total time | 0.860 |
| MFLOPS (average per process) | 34.7 |

Although LAURA displays data cache hit rates and cache line reuse similar to codes such as the NAS Parallel Benchmarks, L1 cache line reuse is a factor of 10 too low for really high performance. However, memory loads contribute a significant amount of time, and LAURA requires several loads to produce a floating-point instruction. LAURA is first and foremost a memory-bound code, one which experiences delays arising from the magnitude of memory requests, rather than a code which performs poorly because of memory access patterns.

## 4. LAURA Optimization

Before the EC diagnostics were available, the initial optimization activity involved examining the large execution-time routines in Table 1 to see if the loops in these subroutines could perform more efficiently. The conventional wisdom regarding the porting of vector codes to RISC computers is that the restructuring of vector loops will reduce cache and TLB misses, thereby increasing performance.

DIRSWP, the subroutine which directs the sweep across the slabs and manages the sliding window, makes many calls to GATFACE throughout the iteration. Since much of the required data is defined at slab centers, GATFACE loads center-defined slab data into face-defined data arrays and these are later converted into face-defined quantities. The loading is accomplished through a double DO-loop which essentially loads several long vectors into several other long vectors. These loops contained little data reuse so cache blocking could not increase the performance of this subroutine.

The situation was the same for the other prominent subroutines: typical cache blocking techniques did not work on LAURA. The code performed most of its array accesses using efficient stride-1 loops with an indexing technique which allowed addressing of two-dimensional arrays with a singe index.

At this point, a discussion with J. Taft about the lack of success with loop-level optimization led to the consideration that elimination of superfluous memory references would the best approach to single-CPU optimization. Since EC data indicated an excessive amount of memory loads and stores as the major performance problem, reduction of these memory references seemed a promising approach.

A printout of subroutine entry and exit points together with the calls made by the DIRSWP routine as it executed the sample problem provided familiarity with the code structure and shed light on the sliding window technique. Large amounts of memory activity appear in DIRSWP both when the code loads and

unloads the center-defined quantities for slabs comprising the sliding window and when the code supplies face-defined quantities prior to computing gradients. It was decided to remove the sliding window scheme first.

Reworking of the memory scheme consisted of changing all references to "in-core" slabs to directly addressing the data from an expanded storage array containing all data, thus reducing the need for calls to the GATSCTA and DROPONE routines. Implementation involved stepping through DIRSWP and resolving the instances in which the modifications produced incorrect results relative to the original problem (simple routines which summed individual common blocks proved to be the cleanest way to discover the erroneous coding). The most difficult aspects of the modification involved the boundary condition arrays because the slab numbers used therein were somewhat implicit.

The careful inspection of the code required by this activity also led to several other modifications:

-DIRSWP always calls GATFACE twice, once for each face, in preparing a slab for calculation: sending parameters of both faces eliminated the second call.
-LIMITER invoked a routine to calculate special minimization function of two vectors: in-lining this call allowed a significant reduction in memory operations.
-DIRSWP followed a call to DRV with a call to ABSEIG to compute the local sound speed and examination of the code indicated that both routines used the same arrays; a modification allowed DRV to execute ABSEIG coding if the input parameters specified ABSEIG execution.

Table 4 shows the dedicated subroutine timings for the Initial and Final versions made on the Hopper Origin machine on 05/29/98. The machine contained 64 250 mHZ R10000 processors each with 256 MB of memory, a 32KB primary cache and a 4 MB secondary cache. The table also shows elapsed-time improvements made to the key subroutines by the optimizations described above. The final version does not call MINMOD because an optimization transferred this calculation to LIMITER and the extra work done by LIMITER explains why it displays an elapsed-time increase in the final version. In the final version, the sum of the elapsed times for MINMOD and LIMITER is about half of their total for the initial version. DROPONE, called several times for each slab by DIRSWP, transferred data from one window location to another and since the optimization removed the window, DROPONE only initializes the data. GATSCTA obtained original data from the storage arrays at the beginning of the slab processing and placed new data into these storage locations at the end of the iteration. The modifications now allow DIRSWP to work on the data directly. GATFACE obtains the slab-centered data for transfer to the slab faces and its elapsed-time improvement arises from combining multiple calls into one call.
.

**Table 4: LAURA Elapsed Times (seconds) for 1000-Step Problem**

| subroutine | Initial Version | Final Version | Improvement |
|---|---|---|---|
| gatface | 230.200 | 204.878 | 25.322 |
| drv | 151.500 | 150.743 | 0.757 |
| invflx | 100.400 | 98.593 | 1.807 |
| dirswp | 97.430 | 100.548 | -3.118 |
| gatgeoa | 94.790 | 94.316 | 0.474 |
| minmod | 94.370 | 0.000 | 94.370 |
| dropone | 74.910 | 0.599 | 74.311 |

**Table 4: LAURA Elapsed Times (seconds) for 1000-Step Problem**

| subroutine | Initial Version | Final Version | Improvement |
|---|---|---|---|
| gatscta | 59.470 | 4.044 | 55.426 |
| limiter | 32.300 | 61.370 | -29.070 |
| total | 1107.865 | 893.465 | 214.400 |

Improvements to the Origin processor may change the overall effect of the memory modifications,. For example, a larger secondary cache size may improve the memory utilization.

## 5. Observations

Removing the sliding window from the LAURA code increases the memory requirements by about 8 MB, although the actual resident size of the program during execution does not reflect the entire increase because the IRIX OS memory segments into pages. Typical large-scale LAURA problems seem to involve increasing the number of processors, while keeping about the same number of points processor. For example, the RLV configuration (without the wake) involved 64 blocks of 48 x 42 x16 [6]. Removal of the sliding window for this case would increase the memory requirements per processor by about the same amount as the current sample problem and would easily fit in the 256 MB memory of the current R10000 processor.

This effort was the first and the most obvious of potential optimizations for LAURA. The large amount of available memory should allow an additional memory-reducing optimization, removing the GATFACE calls and keeping some or all of the face data in core. DIRSWP calls GATFACE to supply slab face values from slab center-defined flow quantities and requires about 20% of LAURA elapsed time.

## 6. Conclusion

The LAURA code is an example of an efficient vector (or legacy) code which employs a windowing technique to limit its overall memory footprint. While RISC machines cannot supply vector-level memory bandwidth for non-local references, they can supply a much greater amount of both local and cache memory per processor and thus perhaps lessen the need for high memory bandwidth. Removal of the sliding window, although conceptually a very simple idea, required a significant effort to contribute to a 15% improvement in floating-point performance. Additional reduction of LAURA memory references, albeit also at an increase in the memory footprint, would be possible through elimination or limitation of the GATFACE calls. Improving the RISC performance of vector codes through memory reference reduction requires little specialized architectural knowledge. The approach does require a good knowledge of the program algorithm, but this knowledge should be available to the typical CFD user.

## 6. References


1. R.J. Bergeron,"The Performance of the NAS HSPs in 2Q93", http://science.nas.nasa.gov/Pubs/TechReports/RNDreports/RND-94-005/RND-94-005.html

2. R.J. Bergeron,"Measurement of a Scientific Workload Using the IBM Performance Monitor", SC98, Orlando Fla., Nov. 8-13, 1998. http://www.supercomp.org/sc98/TechPapers/sc98_FullAbstracts/Bergeron1289/index.htm

3. S. Carr and K. Kennedy, "Compiler Blockability of Numerical Algorithms", Proceedings: Supercomputing'92, Minneapolis, Minnesota, (November 1992), pages 114-124.

4. P. Gnoffo, "An Upwind-Biased, Point-Implicit Relaxation Algorithm for Viscous, Compressible Perfect-Gas Flows," NASA TP-2953, February, 1990.

5. Pulliam, T.H., "Efficient Solution Methods for the Navier-Stokes Equations", Lecture Notes for the Von Karman Institute for Fluid Dynamics Lecture Series: Numerical Techniques for Viscous Flow Computation in Turbomachinery Bladings, Jan. 20-24, 1986, Brussels Belgium.

6. C. J. Riley and F. M. Cheatwood, "Distributed Memory Computing With the Langley Aerothermodynamic Upwind Relaxation Algorithm (LAURA), Fourth NASA National Symposium on Large-Scale Analysis and Design on High-Performance Computers and Workstations, Williamsburg, Virginia, October 15-17, 1997.

7. M. Zagha, et al, "Performance Analysis Using the MIPS R10000 Performance Counters, "Proceedings: Supercomputing'96, Pittsburg, PA, (November 1996).

8. J. Brooks, "Cray Origin 2000 Supercomputer Optimization", 40th Cray User Group Conference, Stuttgart, Germany, June 15-19,1998.